Revisiting the Cache Miss Analysis of Multithreaded Algorithms [⋆]

Richard Cole[1] and Vijaya Ramachandran[2]

[1] Computer Science Dept., Courant Institute, NYU, New York, NY 10012.
[2] Dept. of Computer Science, University of Texas, Austin, TX 78712.

Abstract. This paper revisits the cache miss analysis of algorithms when scheduled using randomized work stealing (RWS) in a parallel environment where processors have private caches. We focus on the effect of task migration on cache miss costs, and in particular, the costs of accessing "hidden" data typically stored on execution stacks (such as the return location for a recursive call).
Prior analyses, with the exception of [1], do not account for such costs, and it is not clear how to extend them to account for these costs. By means of a new analysis, we show that for a variety of basic algorithms these task migration costs are no larger than the costs for the remainder of the computation, and thereby recover existing bounds. We also analyze a number of algorithms implicitly analyzed by [1], namely Scans (including Prefix Sums and Matrix Transposition), Matrix Multiply (the depth n in-place algorithm, the standard 8-way divide and conquer algorithm, and Strassen's algorithm), I-GEP, finding a longest common subsequence, FFT, the SPMS sorting algorithm, list ranking and graph connected components; we obtain sharper bounds in many cases.
While this paper focusses on the RWS scheduler, the bounds we obtain are a function of the number of steals, and thus would apply to any scheduler given bounds on the number of steals it induces.

1 Introduction

Work-stealing is a longstanding technique for distributing work among a collection of processors [4, 17, 2]. Work-stealing operates by organizing the computation in a collection of tasks with each processor managing its currently assigned tasks. Whenever a processor p becomes idle, it selects another processor q and is given (it *steals*) some of q's available tasks. A natural way of selecting q is for p to choose it uniformly at random from among the other processors; we call this randomized work stealing, RWS for short. RWS has been widely implemented, including in Cilk [3], Intel TBB [18] and KAAPI [16]. RWS is also an oblivious scheduler, in that it does not use system parameters such as block and cache size. This methodology is continuing to increase in importance due to its applicability to portable algorithms for multicore computers.

[⋆] Richard Cole (`cole@cs.nyu.edu`) was supported in part by NSF Grant CCF-0830516. Vijaya Ramachandran (`vlr@cs.utexas.edu`) was supported in part by NSF Grant CCF-0830737.

RWS scheduling has been analyzed and shown to provide provably good parallel speed-up for a fairly general class of algorithms [2]. Its cache overhead for private caches was considered in [1], which gave some general bounds on this overhead; these bounds were improved in [15] for a class of computations whose cache complexity function can be bounded by a concave function of the operation count. However, in a parallel execution, a processor executing a stolen task may incur additional cache misses while accessing data on the execution stack of the original task. We will refer to such accesses as "hidden data" accesses. Including these accesses in the cache miss analysis can cause cache miss costs of different tasks performing the same amount of work to vary widely, and this can lead to significant overestimates of the total cache miss cost when using a single bounding concave function.

To capture the cost of hidden data accesses, we provide a largely new analysis. These costs, while not explicitly addressed, are covered by the analysis in [1]. However, the tighter bounds in subsequent work (e.g., [15]) overlook them. Our new analysis accounts for all cache miss costs by identifying a subset of well-behaved tasks, which we term *natural* tasks; our analysis shows that the cache miss costs of HBP algorithms, as defined in Section 2, can be bounded as follows.

Theorem 1. *Consider an execution of an HBP algorithm $\mathcal{A}$ which incurs S steals. Then there is a collection of $s = O(S)$ disjoint natural subtasks $\mathcal{C} = \{\nu_1, \nu_2, \cdots, \nu_s\}$ such that the cache miss cost of $\mathcal{A}$ is bounded by $O(S \log B + \sum_i C(\nu_i))$, where $C(\nu_i)$ is the worst case cost for executing ν_i sequentially starting with an empty cache, and B is the block size. If $\mathcal{A}$ uses linear space, then the $O(S \log B)$ term improves to $O(S)$.*

With this theorem in hand, the analysis of an algorithm reduces to determining its worst case decomposition into disjoint natural tasks and bounding their costs. This tends to be relatively straightforward. Also, the algorithm design task amounts to maximizing the size of a worst-case collection $\mathcal{C}$ for which $\sum_{i \in \mathcal{C}} C(\nu_i)$ has cost bounded by the cache miss cost of a sequential execution; this is a standard parallel algorithm design issue.

1.1 Computation Model

We model a computation using a directed acyclic graph, or dag, D (good overviews can be found in [13, 2]). D is restricted to being a series-parallel graph, where each node in the graph corresponds to a size $O(1)$ computation. Recall that a directed series-parallel graph has start and terminal nodes. It is either a single node, or it is created from two series-parallel graphs, G_1 and G_2, by one of:

i. Sequencing: the terminal node of G_1 is connected to the start node of G_2.

ii. A parallel construct (binary forking): it has a new start node s and a new terminal node t, where s is connected to the start nodes for G_1 and G_2, and their terminal nodes are connected to t.

One way of viewing this is that the computational task represented by graph G decomposes into either a sequence of two subtasks (corresponding to G_1 and

G_2 in (i)) or decomposes into two independent subtasks which could be executed in parallel (corresponding to G_1 and G_2 in (ii)). The parallelism is instantiated by enabling two threads to continue from node s in (ii) above; these threads then recombine into a single thread at the corresponding node t. This multithreading corresponds to a fork-join in a parallel programming language.

We will be considering algorithms expressed in terms of tasks, a simple task being a size $O(1)$ computation, and more complex tasks being built either by sequencing, or by forking, often expressed as recursive subproblems that can be executed in parallel. Such algorithms map to series-parallel computation dags, also known as nested-parallel computation dags.

In RWS, each processor maintains a work queue, on which it stores tasks that can be stolen. An idle processor C' picks a processor C'' uniformly at random and independently of other idle processors, and attempts to take a task (to steal) from the top of C'''s task queue. If the steal fails (either because the task queue is empty, or because some other processor was attempting the same steal, and succeeded) then processor C' continues trying to steal, continuing until it succeeds.

We consider a computing environment which comprises p processors, each equipped with a local memory or cache of size M. There is also a shared memory of unbounded size. Data is transferred between the shared and local memories in size B blocks (or cache lines). The term *cache miss* denotes a read of a block from shared-memory into processor C's cache, when a needed data item is not currently in cache, either because the block was never read by processor C, or because it had been evicted from C's cache to make room for new data.

There is another cost that could arise, namely cache misses due to false sharing. As in [1, 15], we assume that there is no false sharing, perhaps as a result of using the Backer protocol, as implemented in Cilk [3]. Even when false sharing is present [10, 12], the cache miss costs as analyzed here remain relevant, since false sharing can only further increase the costs.

Execution Stacks. Now we explain where the variables generated during the computation are stored, including the variables needed for synchronization at joins and for managing procedure calls. In a single processor algorithm a standard solution is to use an execution stack. We proceed in the same way, with one stack per thread. Before elaborating we define task kernels.

Definition 1. *A task kernel is the portion of a task computation dag that remains after the computation dags for all its stolen subtasks are removed.*

The original task in the algorithm and each stolen subtask will have a separate computation thread. The work performed by a computation thread for a task τ is to execute the task kernel τ^K for task τ. Each computation thread will keep an execution stack on which it stores the variables it creates: variables are added to the top of the stack when a subtask begins and are released when it ends.

Usurpations. Let C be the processor executing task kernel τ^K. As τ^K's execution proceeds, the processor executing it may change. This change will occur at a join node v at which a stolen subtask τ' ends, if the processor C' that was executing τ' reaches the join node later than C. Then, C' continues the execution of τ^K going forward from node v. C' is said to *usurp* the computation of τ^K; we also say that C' usurps C. In turn, C' may be usurped by another processor C''. Indeed, if there are k steals of tasks from τ, then there could be up to k usurpations during the computation of τ^K.

A usurpation may cause cache misses to occur due to hidden data. By hidden data we mean undeclared variables stored on a task's execution stack such as the variables used to control task initiation and termination. If in cache, this data can be accessed for free by a processor C, but a usurper C' incurs cache misses in accessing the same data.

1.2 Prior Work

In both [1] and [15] the computation is viewed as being split into subtasks both at each fork at which there is a steal and at the corresponding join node, and then the costs of these subtasks are bounded by making a worst case assumption that each subtask is executed beginning with an empty cache. Further, in [1] and for most analyses in [15], blocks are assumed to have size $O(1)$. In [1], the following simple observation is made: whether or not it is stolen, a subtask accessing $2M$ or more words would incur the same number of cache misses, up to constant factors. Thus the cache miss overhead due to the steals is bounded by $O(M \cdot S)$, where S is the number of stolen tasks. In [15], improved bounds are obtained in the case the cache misses incurred by any task in the computation can be bounded by a concave function C_f of the work the task performs; if W is the total work, the cache miss cost is bounded by $S \cdot C_f(W/S)$, which can yield better bounds when the average stolen task size is less than M.

1.3 Our Results

We use the following parameters to specify our results. Let D be the computation dag of a multithreaded algorithm $\mathcal{A}$. Let n be the input size. Suppose that an operation on in-cache data takes $O(1)$ time units, that the cost of a cache miss is $O(b)$ time units, that the cost for an attempted steal of a task, successful or not, is $\Theta(c_s)$ time units [3]. We will assume that $b = O(c_s)$, which seems reasonable as each successful steal entails at least one cache miss.

The following bound on the runtime of a parallel algorithm $\mathcal{A}$ scheduled using RWS indicates its dependence on the cache miss analysis. Suppose that $\mathcal{A}$ performs W operations in the worst case. Let S be the number of stolen tasks, let $C(S, B)$ denote an upper bound on the number of cache misses incurred in a parallel execution with S steals, and let $U(p)$ denote the cost of unsuccessful steals. Then, since a processor is either computing, accessing data or attempting

[3] [10] shows how to generalize the analysis in [1] bounding the number of steals so as to allow unsuccessful steals to take just $O(c_s)$ time units (instead of $\Theta(c_s)$).

Algorithm	**Q**	**$C(S,B)$**	
		In [1]	**Our Results**
Scans, Matrix Transpose (MT)	$\frac{n}{B}$	$Q + M \cdot S$	$Q + S$
Depth-n-MM, 8-way MM, Strassen, I-GEP	$n^3/(B\sqrt{M})$	$Q + M \cdot S$	$Q + S^{\frac{1}{3}} \frac{n^2}{B} + S$ (in BI) [4] $Q + S^{\frac{1}{3}} \frac{n^2}{B} + S \cdot B$ (in RM)
Finding LCS sequence	$n^2/(BM)$	$Q + M \cdot S$	$Q + n\sqrt{S} + S$ [4]
FFT, SPMS Sort	$\frac{n}{B} \log_M n$	$Q + M \cdot S$	$Q + S \cdot B + \frac{n}{B} \frac{\log n}{\log[(n \log n)/S]}$
List Rank., Graph Connected Comp.	See the full paper		

Table 1. Bounds for cache miss overhead, $C(S,B)$, under RWS in [1] (column 3, with $B = O(1)$) and our results (column 4) for some HBP algorithms; $O(\ \cdot\ )$ omitted on every term. The sequential cache complexity is Q (a term $f(r)$, specified below in Definition 5, is omitted from Q). Always, the new bound matches or improves the bound in [1].

to steal at each time step, $\mathcal{A}$ runs in time

$$O\left(\frac{1}{p} \cdot (W + b \cdot C(S,B) + c_s \cdot S + U(p))\right).$$

For HBP algorithms, the term $U(p)$ is subsumed by the other terms [10].

Table 1 gives our bounds on C. Previous work in [15] obtained the bounds for $C(S,B)$ shown under Our Results in Table 1 for the Depth-n Matrix Multiply [14], I-GEP [6] and computing the length of an LCS [5, 7] (and some stencil computations); as already noted, these bounds overlooked some costs, now included. We determine new bounds on $C(S,B)$ for several other algorithms (FFT, SPMS sort, List Ranking, Connected Components, and others). [1] had obtained a bound of $O(S \cdot M)$ for every algorithm, assuming $B = O(1)$.

Road-map: In Section 2 we review the definition of HBP algorithms. In Section 3 we bound the cache miss costs for BP algorithms, a subset of HBP algorithms; the analysis for HBP algorithms is deferred to the full paper for lack of space. In Section 4, we then apply this analysis to obtain the cache miss bounds for FFT; the remaining results are in the full paper.

2 HBP Algorithms

We review the definition of HBP algorithms [11, 9]. Here we define the size of a task τ, denoted $|\tau|$, to be the number of already declared distinct variables it accesses over the course of its execution (this does not includes variables τ declares during its computation). Also, we will repeatedly use the following

[4] These bounds were obtained for Depth-n-MM and I-GEP by [15], but with hidden costs overlooked, though they do not change the overall bound. For LCS, [15] bounded the cost of finding the length of the sequence, but not the sequence itself.

notation: τ_w will denote the steal-free task that begins at a node w in a fork tree, and ends at the corresponding node in the corresponding join tree.

Definition 2. *A* BP computation π *is an algorithm that is formed from the down-pass of a binary forking computation tree T followed by its up-pass, and satisfies the following properties.*

i. In the down-pass, a task that is not a leaf performs only $O(1)$ computation before it forks its two children. Likewise, in the up-pass each task performs only $O(1)$ computation after the completion of its forked subtasks. Finally, each leaf node performs $O(1)$ computation.

ii. Each node declares at most $O(1)$ variables, called local variables*; π may also use size $O(|T|)$ arrays for its input and output, called* global variables*.*

iii. Balance Condition. *Let w be a node in the down-pass tree and let v be a child of w. There is a constant $0 < \alpha < 1$ such that $|\tau_v| \leq \alpha|\tau_w|$.*

A BP computation can involve sharing of data between the tasks at the two sibling nodes in the down-pass tree. However, it is not difficult to see that the size k of a BP computation (i.e., the number of nodes in the down-pass or up-pass tree) is polynomial in the size n of the task at the root of the BP computation. As it happens, for all the BP algorithms we consider, $k = \Theta(n)$. A simple BP example is the natural balanced-tree procedure to compute the sum of n integers. This BP computation has $n = \Theta(k)$, and there is no sharing of data between tasks initiated at sibling nodes in the down-pass tree.

Definition 3. *A* Hierarchical Balanced Parallel (HBP) Computation *is one of the following:*

1. A Type 0 Algorithm, a sequential computation of constant size.

2. A Type 1 Algorithm, a BP computation.

3. Sequencing. A sequenced HBP algorithm of Type t results when $O(1)$ HBP algorithms are called in sequence, where these algorithms are created by rules 1, 2, or 4, and where t is the maximum type of any HBP algorithm in the sequence.
4. Recursion. A Type $t+1$ recursive *HBP algorithm, for $t \geq 1$, results if, on an input of size n, it calls, in succession, a sequence of $c = O(1)$ ordered collections of $v(n) \geq 1$ parallel recursive subproblems, where each subproblem has size $\Theta(r(n))$, where $r(n)$ is bounded by αn for some constant $0 < \alpha < 1$.*

Each of the c collections can be preceded and/or followed by a sequenced HBP algorithm of type $t' \leq t$ and at least one of these calls is of type exactly t. If there are no such calls, then the algorithm is of Type 2 if $c \geq 2$, and is Type 1 (BP) if $c = 1$. Each collection of parallel recursive subproblems is organized in a BP-like tree T_f, whose root represents all of the $v(n)$ recursive subproblems, with each leaf containing one of the $v(n)$ recursive subproblems. In addition, we require the same balance condition as for BP computations for nodes in the fork tree.

Lemma 1. *Let u and v be the children of a fork node. Then $|\tau_u| = \Theta(|\tau_v|)$.*

Proof. Let w denote the fork node. $|\tau_w| \leq 1+|\tau_u|+|\tau_v|$, since only $O(1)$ variables are accessed by the computation at node w. As $|\tau_v| \leq \alpha|\tau_w|$, $|\tau_u| \geq (1-\alpha)|\tau_w|-1$, and hence $|\tau_u| \geq \frac{1-\alpha}{\alpha}|\tau_v| - O(1) = \Theta(|\tau_v|)$.

Matrix Multiply (MM) with 8-way recursion is an example of a Type 2 HBP algorithm. The algorithm, given as input two $n \times n$ matrices to multiply, makes 8 recursive calls in parallel to subproblems with size $n/2 \times n/2$ matrices. This recursive computation is followed by 4 matrix additions, which are BP computations. Here $c = 1$, $v(n^2) = 8$, and $r(n^2) = n^2/4$. Depth-n-MM [14, 8] is another Type 2 HBP algorithm for MM with $c = 2$, $v(n^2) = 4$, and $r(n^2) = n^2/4$.

Linear Space Bound. We obtain stronger bounds for computations that are linear space bounded. Linear space boundedness simply means that the computation uses space that is linear (or smaller) in the size of its input and output data, with the additional caveat that in an HBP computation, the linear space bound also applies to all recursive tasks.

All the algorithms analyzed in this paper are linear space bounded.

Constraint on Accesses to Variables. In order to control cache misses when a usurpation occurs we limit accesses which are made to local variables (variables a procedure declares) as follows:
The computation at node v' in an up-pass tree may access global variables and those local variables declared either in the corresponding node v in the down-pass tree or at v's parent, but no others; note that no variables would be declared at node v' for it ends a subcomputation. (While it is natural to limit accesses in the down-pass in the same way, it is not necessary for our results.)

With the HBP definition in hand, we can now define natural tasks.

Definition 4. *A* Natural Task *is one of the following:*

1. A task built by one of rules 1–4 in Definition 3.

2. A task that could be stolen: a task τ_w beginning at a node w in a fork tree and ending at the corresponding node of the corresponding join tree and including all the intermediate computation.

Work Stealing Detail. At a fork node, it is always the right child task that is made available for stealing by being placed on the bottom of the task queue. This ensures that in a BP computation, a task kernel τ^K always comprises a contiguous portion of the leaves in τ, thereby minimizing cache misses. For an HBP computation, an analogous requirement applies to each ordered collection of parallel recursive tasks. As is standard in work stealing, steals are performed at the top of the task queue, which is a double-ended queue.

3 Bounding the Cache Misses

Some computations such as those on matrices in row major format incur extra cache miss costs when there is need to access a collection of data that are not packed into contiguous locations. The analysis of such accesses has been studied widely in sequential cache efficient algorithms (see, e.g., [14]); the notion of a 'tall cache' has often been used in this context. In work-stealing this issue is more challenging, since there is less control over organizing the sequencing of tasks to minimize these costs. To help bound the cache miss costs of non-constant sized blocks, we formalize the notion of data locality with the following definition.

Definition 5. *A collection of r words of data is f-cache friendly if they are contained in $O(r/B + f(r))$ blocks.*

The data accessed by the tasks τ in the algorithms we consider are all either $O(1)$- or $O(\sqrt{|\tau|})$-friendly.

We limit the analysis to well-behaved functions f, which we call *regular* f.

Definition 6. *A cache-friendliness function f is* regular *if it is a non-decreasing polynomially bounded function.*

Next, we review the primary problem analyzed in [15] to show why the approach using concave functions need not yield tight bounds.

Depth-n matrix multiply. Possibly there is a steal of a leaf node task τ_v in a final recursive call; if the processor P' executing τ_v usurps the remainder of the computation, then P' is going to carry out the processing on the path to the root of the up-pass tree, and P' could end up executing $\log n$ nodes in the up-pass. All that has to be done at these nodes is to synchronize and possibly terminate a recursive call, but these both require accessing a variable stored on the execution stack E_τ for the parent task τ (the one from which the steal occurred). The variables P' accesses will be consecutive on E_τ. This results in $\Theta([\log n]/B)$ cache misses. In general, a task that performs $x \geq \log n$ work may incur $\Theta(x/[B\sqrt{M}] + [\log n]/B)$ cache misses which yields a bound of $\Theta(S^{\frac{1}{3}} \frac{n^2}{B} + [S \log n]/B)$ cache misses rather than $O(S^{\frac{1}{3}} \frac{n^2}{B} + S)$ as in [15]; the former is a larger bound when $B = o(\log n)$ and $S \geq n^3/\log^{3/2} n$.

Our Approach. Our method determines bounds on the worst case number of natural tasks of a given size, and shows that the cache miss cost of the given algorithm is bounded by $\Theta(S)$ plus the costs of $O(S)$ disjoint natural tasks. For tasks τ of size $2M$ or larger, the same cache miss costs would be incurred, up to constant factors, even if there were no steal, modulo a term $O(f|\tau|)$. For smaller tasks, the incurred costs are a function of the task size, which combined with the bounds on the number of tasks of a given size, yields bounds on the cache miss costs as a function of the number of stolen tasks. More specifically, we prove our bound in three parts:

1. A bound assuming there are no usurpations.

2. A bound on the cost of the up-passes following the usurpations.

3. A bound on the costs of re-accessing data following a usurpation, aside the costs in (2).

For BP computations, which we analyze in the next section, (3) does not arise; (1) is handled by Lemma 2 below, and (2) by Lemma 3. For HBP computations the argument is more involved; it is given in the full paper. The analysis in [15] bounds (1) and (3) accurately, assuming the cache miss function is a tight concave function of the work, but it does not bound (2).

3.1 Analysis of BP Computations

Let $C(\tau)$ denote the worst case cache miss cost for sequentially executing τ starting with an empty cache. Let T_w denote the subtree of the down-pass tree rooted at node w. Also, we define a node w to be *steal-free* if its right child is not stolen (recall that only w's right child could be stolen). Analogously, we say T_x is steal-free if all its nodes are steal-free.

Definition 7. *An execution of task kernel τ^K is* usurpation-free *if at every join ending a steal the processor C currently executing τ^K continues to execute τ^K following the join.*

Lemma 2. *Let τ^K be a task kernel, and let the cache friendliness function f be regular. Suppose that τ incurs $S \geq 1$ steals in forming τ^K, and suppose that τ^K's execution is usurpation-free. Then there is a natural task τ_u fully contained in τ^K such that the execution of τ^K starting with an empty cache incurs $O(S + C(\tau_u))$ cache misses.*

Proof. Let w be the first steal-free node (a node whose child is not stolen) on the path P starting at τ's root (its start node) and descending to the left. Then we define node u as follows. If w is a non-leaf node of the down-pass tree, then u is the left child of w. Otherwise, u is node w itself. τ_u denotes the natural task that begins at node u and ends at the corresponding node in the up-pass tree. Let v denote u's sibling.

τ^K incurs at most the following number of cache misses: $O(|P| + C(\tau_w))$, since τ^K completes an initial portion of τ_w, followed by the $O(1)$ computation at each node in P. But τ_w incurs at most $O(1+C(\tau_u)+C(\tau_v))$ cache misses, and $C(\tau_v) = O(C(\tau_u))$, as $|\tau_v| = O(|\tau_u|)$ by Lemma 1, and by the regularity condition on $f(\tau)$. Also, $|P| = O(S)$. Thus τ^K incurs $O(S + C(\tau_u))$ cache misses.

Comment. We note that Lemma 2 implies that the analysis in [15] applies to usurpation-free BP computations.

Lemma 3 bounds the additional costs in BP computations due to usurpations, namely the costs for executing the nodes on the *usurpation path*, the path from the first node at which a usurpation occurs up to the root of the up-pass tree.

Lemma 3. *Let τ^K be a task kernel and let the cache friendliness function f be regular. Suppose that τ incurs $S \geq 1$ steals in forming τ^K. Then there is a natural task τ_u fully contained in τ^K such that executing τ^K starting with an empty stack incurs $O(C(\tau_u) + S)$ cache misses.*

Proof. w, u, v and τ_u are defined as in the proof of Lemma 2.

To bound the cost of the usurpation path we partition it into alternating subpaths heading up to the right and up to the left. Let P'_{R_i}, $1 \le i \le s$, denote the paths of nodes heading up to the right, P'_{L_i}, $1 \le i \le s'$, be the paths heading up to the left. Note that $s - 1 \le s' \le s + 1$, and that $s \le S$.

We explain where the steals from τ^K occur in terms of these paths. Let x' be a node on a path P'_{R_i} and let x be the corresponding node in the down-pass tree. Then x's right child is stolen, whereas the nodes y corresponding to nodes y' on the paths P'_{L_i} are steal-free.

Usurpations occur only at nodes x' on the paths P'_{R_i}. Following a usurpation at a node x' (which we call a *usurped* node) the remaining work is to traverse the path from x' toward the next usurpation site, or to the root of the up-pass tree, if this is the final usurpation of τ^K. There are at most $O(1)$ cache misses for each node traversed, but the bound for the paths P'_{L_i} can be smaller.

Traversing the paths P'_{R_i} incurs $O(\sum_i |P'_{R_i}|) = O(S)$ cache misses.

In traversing the paths P'_{L_i} there are two possible costs: costs for accessing global variables and costs for accessing local variables, variables stored on τ^K's execution stack. We analyze each in turn.

For each node on a path P'_{L_i} there could be $O(1)$ accesses to global variables. Let w'_i be the node on P'_{L_i} that is closest to the root, let u'_i and v'_i be its left and right child, respectively, and let u_i and v_i be the corresponding nodes in the down-pass tree. Thus, all nodes in the subtrees rooted at u_i and u'_i are in τ^K, and by the BP balance property, $|\tau_{u_i}| = \Omega(|\tau_{v_i}|)$. Since, for $1 \le i \le s'$, τ_{v_i} includes all nodes in P'_{L_i} except for w'_i, and as $O(1)$ data is accessed at w'_i, this implies that the global accesses for nodes on P'_{L_i} cost $O(C(\tau_{u_i}))$. Finally, since the task sizes decrease geometrically as we descend a BP tree, the sum of these sizes is dominated by $C(\tau_u)$, since $u_{s'} = u$ by its definition in Lemma 2.

Also, there are $O(1)$ accesses to τ^K's execution stack at each node x' on P'_{L_i}. These accesses are to the $O(1)$ variables for the corresponding node x in the down-pass tree, or for x's parent. For each path P'_{L_i} the accesses cost $O(\lceil |P'_{L_i}|/B \rceil)$, since the variables for successive nodes are stored consecutively. Now $\sum_i |P'_{L_i}|/B = O(\text{ht}(T_w)/B) = O([\log|\tau_w]/B)$ (the final inequality follows from the definition of BP computations). Thus $O(\sum \lceil |P'_{L_i}|/B \rceil) = O(\lceil \log|\tau_w|/B \rceil)$ $= O(\lceil [\log|\tau_u| + 1]/B \rceil) = O(\lceil |\tau_u|/B \rceil) = O(C(\tau_u))$.

Overall, this totals $O(C(\tau_u) + S))$ cache misses.

Theorem 2. *Let $\mathcal{A}$ be a BP algorithm and suppose that its cache friendliness function f is regular. Consider an execution of $\mathcal{A}$ which incurs $S \ge 1$ steals. Then there is a collection $\mathcal{C} = \{\nu_1, \nu_2, \cdots \nu_s\}$ of $s = S + 1$ disjoint natural tasks such that the execution of $\mathcal{A}$ incurs at most $O(S + \sum_i C(\nu_i))$ cache misses.*

Proof. Let $\tau_1, \tau_2, \cdots, \tau_{S+1}$ denote the original task and the S stolen tasks in the execution of $\mathcal{A}$. A collection $\mathcal{C}$ is constructed by applying Lemma 3 to each of the $s' \le S$ tasks τ_i that incur a steal, and adding the s' tasks it identifies to $\mathcal{C}$. In addition, for the remaining $S + 1 - s'$ tasks τ_i that do not incur a steal, τ_i itself is added to $\mathcal{C}$. Thus $|\mathcal{C}| = S + 1$.

HBP Algorithms. The above analysis can be extended to type $t \geq 1$ HBP algorithms, leading to Theorem 1. Its proof is based on showing a bound analogous to Lemma 2 for natural HBP tasks τ; the bound will now be of the form $O(s \cdot t \log B + \sum_{i \in \mathcal{C}} C(\nu_i))$, where s is the number of steals τ incurs and $\mathcal{C} = O(s \cdot t)$ (we consider t to be a constant). The argument is inductive, the building blocks being BP computations and fork trees incurring steals. We then combine these units hierarchically into natural tasks, following the structure of the HBP computation. Let the HBP computation make a sequence of $c \geq 1$ calls to ordered collections of parallel recursive tasks. Then, each combination will require the addition of a further $O(c)$ tasks to $\mathcal{C}$, and either brings together two sets of steals, or increases the type of the resulting unit, and thereby yields the bound stated in Theorem 1. The analysis is tighter when the computation is linear space bounded, reducing the term $s \cdot t \cdot \log B$ to $s \cdot t$, and viewing t to be a constant, this results in the two versions of the bound in the theorem.

4 Analysis of FFT

We analyze the FFT algorithm described in [14, 8]. The algorithm views the input as a square matrix, which it transposes, then performs a sequence of two recursive FFT computations on independent parallel subproblems of size $\sqrt{n}$, and finally performs a matrix transpose (MT) on the result. This algorithm has sequential cache complexity $Q = O(\frac{n}{B} \log_M n)$ [14].

The Type 2 HBP algorithm FFT, when called on an input of length n, makes a sequence of $c = 2$ calls to FFT on $v(n) = \sqrt{n}$ subproblems of size $r(n) = \sqrt{n}$ with a constant number of BP computations of MT performed before and after each collection of recursive calls. It has $f(r) = \sqrt{r}$.

Lemma 4. *The FFT algorithm incurs* $O(\frac{n}{B} \log_M n + S \cdot B + \frac{n}{B} \frac{\log n}{\log[(n \log n)/S]})$ *cache misses when it undergoes* S *steals.*

Proof. We apply Theorem 1. Each task ν_i in $\mathcal{C}$ incurs at most $|\nu_i|/B + f(|\nu_i|)$ more cache misses than would occur in its execution as part of a sequential execution of the algorithm. As the sequential execution incurs $O((n/B) \log_M n)$ cache misses, it follows that $\sum_i C(\nu_i) = O((n/B) \log_M n + \sum_i |\nu_i|/B + f(|\nu_i|))$. For FFT, $f(r) = O(\sqrt{r})$; thus if $|\nu_i| < B^2$ then $f(|\nu_i|) = O(B)$, and if $|\nu_i| \geq B^2$, then $f(|\nu_i|) = O(|\nu_i|/B)$; so $\sum_i f(|\nu_i|) = O(S \cdot B + \sum_i |\nu_i|/B)$.

It remains to bound the term $\sum_{\nu_i \in \mathcal{C}} |\nu_i|/B$. The total size of tasks of size r or larger is $O(n \log_r n)$, and there are $\Theta(\frac{n}{r} \log_r n)$ such tasks. Choosing r so that $S = \Theta(\frac{n}{r} \log_r n)$, implies that $r \log r = \Theta(n \log n/S)$, so $\log r = \Theta(\log([n \log n/S])$. Thus $\max_{\mathcal{C}} \sum_{\nu_i \in \mathcal{C}} \frac{|\nu_i|}{B} = O(\frac{n}{B} \log_r n) = O(\frac{n}{B} \frac{\log n}{\log[(n \log n)/S]})$.

As shown in the full paper, this yields linear (optimal) speedup for $n \geq p \log\log n \cdot (M^\epsilon + B^2 \log M)$ for any fixed $\epsilon > 0$, This improves on the bound of $n \geq pMT_\infty$ in [1], which requires $n \geq p \log\log n \cdot BM \log M$ for optimal speed-up.

Acknowledgements. We thank the reviewers for their thoughtful suggestions.

References

1. U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002. Springer.
2. R. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, pages 720–748, 1999.
3. R. D. Blumofe, C. F. Joerg, B. C. Kuzmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, 1995.
4. F. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 187–194, 1981.
5. R. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *Proc. of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '06, pages 591–600, 2006.
6. R. Chowdhury and V. Ramachandran. The cache-oblivious Gaussian Elimination Paradigm: Theoretical framework, parallelization and experimental evaluation. *Theory of Comput. Syst.*, 47(1):878–919, 2010.
7. R. A. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *Proc. of the Twentieth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 207–216, 2008.
8. R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. In *Proc. 2010 IEEE International Symposium on Parallel & Distributed Processing*, IPDPS '10, pages 1–12, 2010.
9. R. Cole and V. Ramachandran. Resource oblivious sorting on multicores. In *Proc. of the Thirty Seventh International Colloquium on Automata, Languages and Programming*, ICALP'10, pages 226–237. Springer-Verlag, 2010.
10. R. Cole and V. Ramachandran. Analysis of randomized work stealing with false sharing. *CoRR*, abs/1103.4142, 2011.
11. R. Cole and V. Ramachandran. Efficient resource oblivious algorithms for multicores. *CoRR*, abs/1103.4071, 2011.
12. R. Cole and V. Ramachandran. Efficient resource oblivious algorithms for multicores with false sharing. In *Proc. IEEE IPDPS*, 2012. To appear.
13. T. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* MIT Press, 2009.
14. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. Fortieth Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–297, 1999.
15. M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. *Theory Comput Syst*, 45:203–233, 2009.
16. T. Gautier, X. Besseron, and L. Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proc. International Workshop on Parallel Symbolic Computation*, PASCO '07, pages 15–23, 2007.
17. R. H. J. Halstead. Implementation of Multilistp: Lisp on a multiprocessor. In *Proc. ACM Symposium on LISP and Functional Programming*, pages 9–17, 1984.
18. A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in tbb. In *Proc. IEEE International Symposium on Parallel and Distributed Processing*, IPDPS '08, pages 1–8, 2008.